FORSCHUNGSZENTRUM JÜLICH GmbH
Zentralinstitut für Angewandte Mathematik
D-52425 Jülich, Tel. (02461) 61-6402

Interner Bericht

SVM-Fortran
Reference Manual
Version 1.4

Rudolf Berrendorf, Michael Gerndt

KFA-ZAM-IB-9510

April 1995
(Stand 19.04.95)

SVM-Fortran
Reference Manual *
Version 1.4

Rudolf Berrendorf, Michael Gerndt

Zentralinstitut für Angewandte Mathematik
Forschungszentrum Jülich
D-52425 Jülich

{r.berrendorf, m.gerndt}@kfa-juelich.de

April 24, 1995

*The work described in this report is being carried out as a part of the Esprit project "Performance-Critical Applications of Parallel Architectures (APPARC) " and of the KFA-Intel collaboration.

Contents

1 Introduction

SVM-Fortran is an extension to Fortran77 intended to be used to program highly parallel systems with a global address space. Motivation for the definition of the language have been the lack of support to program massively parallel machines with shared virtual memory. Guidelines for designing the SVM-Fortran language have been:

- provide a base set of language features to express parallel programs for massively parallel computers with a global address space
- support both data parallelism and functional parallelism
- support incremental parallelization for large programs
- allow future language extensions in an easy way
- don't restrict the underlying programming model to Fortran (although this is the primary language we look for)

2 Syntactic Conventions

All SVM-Fortran language constructs are specified as directives beginning with `CSVM$` or as calls to library routines. Directives may be continued on following lines by writing `CSVM$` followed by any non-blank character in column 6.

In the grammar given for SVM-Fortran *xyz-list* denotes a comma-separated list of items *xyz*. The symbols [ ] are part of the meta-language and enclose optional items.

In the following we will distinguish between the following verbs:

- **shall** expresses a requirement
- **should** expresses a recommendation
- **may** expresses a permission
- **can** expresses a possibility

All directives defined by SVM-Fortran are given below. We will omit the preceding `CSVM$`-part of a directive in the following. The directives are split into directives which may occur in the declaration part of a program unit (*declaration-directive*) and directives which may occur in the execution part of a program unit (*execution-directive*). Additionally, intrinsic functions (*intrinsics-call*) callable as any other function will be given in a later chapter.

Syntax:

svmf-directive	**is**	*declaration-directive*
	or	*execution-directive*
declaration-directive	**is**	*default-directive*
	or	*shared-directive*
	or	*private-directive*
	or	*processor-directive*
	or	*template-directive*
	or	*distribute-directive*
execution-directive	**is**	*copy-directive*
	or	*parloop-directive*
	or	*psection-directive*
	or	*repregion-directive*
	or	*exclregion-directive*
	or	*coordinated-call-directive*
	or	*barrier-directive*
	or	*critical-section-directive*
	or	*lock-directive*
	or	*atomic-update-directive*
	or	*create-directive*
	or	*destroy-directive*
	or	*undef-directive*
	or	*redistribute-directive*
	or	*prefetch-directive*

◇

3 Execution Model

Before we start to discuss the execution model, let us define some notations. Some names which are used but not defined here will be defined in one of the following chapters.

A program region is a textual part of a program unit. Program regions may be nested. They are defined as follows:

1. A program unit is a program region.
2. The body of a parallel loop is a program region.
3. A section of a parallel section is a program region.
4. An exclusive region is a program region.
5. A replicated region is a program region.

A parallel program region is a program region which is defined as follows:

1. The body of a parallel loop is a parallel program region.
2. A section of a parallel section is a parallel program region.

Example: Program regions

```
C>>>  here starts a program region
      PROGRAM program_region
      REAL a(100)
CSVM$ PRIVATE:: i

CSVM$ PDO
      DO i=1,100
C>>>  here starts a new (parallel) program region
         a(i) = i
C<<<  here ends the (parallel) program region
      ENDDO
      STOP
      END
C<<<  here ends a program region
```

■

A parallel construct is one of the following:

1. A parallel loop is a parallel construct.

2. A parallel section is a parallel construct.

A coordinated construct is one of the following:

1. A parallel loop is a coordinated construct.
2. A parallel section is a coordinated construct.
3. An exclusive region is a coordinated construct.
4. A replicated region is a coordinated construct.
5. A coordinated call is a coordinated construct.
6. A redistribute directive is a coordinated construct.
7. A create directive is a coordinated construct.
8. A destroy directive is a coordinated construct.
9. An undef directive is a coordinated construct.

3.1 Processors and Processor Sets

A program is executed by abstract processors. There shall be a one-to-one correspondence between abstract processors and physical processors of the computer.

SVM-Fortran has the concept of a processor set which consists of a number of abstract processors. Which processors belong to a processor set is determined at the start of the program, on entering a parallel construct, or on calling a subprogram in replicated mode. The processor set working on a program region is called the active processor set (APS) with respect to that program region.

All processors assigned to the application are members of the initial processor set which is also the active processor set of the main program.

On entering a parallel construct, the active processor set may get split up into several disjoint processor sets which become active on a parallel program region of the parallel construct, e.g. one processor set for each section of a parallel section. These new processor sets are called derived processor sets, derived from the original processor set. The processors in the derived processor sets are ordered with respect to the original processor set. After the derived processor sets have finished their work, these sets are deleted and the old active processor set gets active again. How a processor set is split into derived processor sets is system dependent unless the partitioning is managed by the programmer, e.g. with loop scheduling.

Processors in a processor set are numbered from 0 to n-1 (for n processors belonging to the processor set). It is possible for a processor to get this number relative to the active processor set and relative to the initial processor set.

Example: Processor Sets

```
        PROGRAM processor_set
C***    point 1
        count = 1

CSVM$   PSECTION
CSVM$   SECTION
C***    point 2
        DO i=1,n
           ix(i) = i
        ENDDO
CSVM$   SECTION
C***    point 3
        DO i=1,m
           iy(i) = i
        ENDDO
CSVM$   PSECTION_END

C***    point 4
        END
```

Assume, that 2 processors belong to the initial processor set (ps0) which is also the active processor set at point 1. The parallel section is a new parallel program region. The processor

set ps0 is split up into 2 derived processor sets called ps1 and ps2. Each derived processor set works on one section, i.e. processor set ps1 reaches point 2 and processor set ps2 reaches point 3. After the parallel section (point 4) processor set ps0 gets active again. ■

3.2 Execution Mode

Program regions are executed in either of two execution modes:

1. In the exclusive mode one processor of the active processor set (master processor; `MYPROCSET( )` = 0) executes the code while the other processors of the active processor set are suspended until they are reactivated on a coordinated construct.

2. In the replicated mode, all processors of the active processor set work in parallel and independent of each other on the code, e.g. eventually having a different flow of control.

   Unless a coordinated call is specified, subprogram calls in a replicated region are executed independently by every processor itself. Every processor reaching such a subprogram call forms a derived processor set consisting of that processor only. After execution of that subprogram, the old processor set gets active again.

   On a coordinated construct the processors behave as follows:

   - On a parallel loop or parallel section the work is shared among the processors of the active processor set.
   - On a coordinated call the active processor set remains unchanged (different to non-coordinated calls; see above) and all processors of the active processor set enter the subprogram.
   - On an exclusive region the execution mode is switched to exclusive and all processors but the master processor are suspended.
   - Template operations which are coordinated constructs are done in a synchronous fashion.

Execution of an application starts in the execution mode which applies to the main program. The default execution mode is the exclusive mode and can be changed for a region of code (see Section 5.3). Parallel loop iterations and parallel sections are executed in the execution mode of the enclosing directive which determines the execution mode or the default if no such directive exists. The execution mode can be determined statically for every statement of a program unit. Using the exclusive mode as the default has the advantage that compiling a sequential program with an SVM-Fortran-compiler behaves as compiled with the ordinary sequential compiler (which does not apply for the replicated mode).

Example: Calling a subroutine with a coordinated call

```
        PROGRAM subtest
        DIMENSION a(100), component_sum(100)
CSVM$ PRIVATE:: i
CSVM$ PROCESSORS:: P(4)

CSVM$   REPLICATED_REGION
           i = myprocset() * 25 + 1
C***       executed by every processor independently
```

```
C***        in derived processor sets
            CALL sub(a(i),b(i),component_sum(i),25)

C***        executed by all processors with a coordinated call
CSVM$       COORDINATED_CALL
            CALL sub(a,b,sum,100)
CSVM$    REPLICATED_REGION_END
         END

         SUBROUTINE sub(a,b,res,n)
         DIMENSION a(n),b(n),res(n)
CSVM$    PRIVATE:: i
CSVM$    PDO(REDUCTION(res))
         DO i=1,n
           res = res + a(i) + b(i)
         ENDDO
         RETURN
         END
```

Assume, the program is executed by 4 processors. The replicated region in the main program is executed by all processors in the active processor set. The first call to subroutine *sub* is executed by every processor of the active processor set independently. Every derived processor set consists of one processor. In the subroutine *sub*, the parallel loop is therefore executed `NUMPROC()` times, each loop with 25 iterations executed by one processor (as each derived processor set consists of exactly one processor). The second call to *sub* is executed by all processors of the active processor set, the active processor set remains. The parallel loop in *sub* is executed once, all processors share the work on n=100 iterations. ■

4 Data Sharing

By default, the usual Fortran memory model applies to all variables, i.e. storage association (e.g. the mapping of multi-dimensional arrays to a one-dimensional array) and sequence association (e.g. the storage relationship of objects in a common block). How data is distributed initially over processors (e.g. pages of the virtual memory system) is implementation dependent. It is not possible to distribute data explicitly across processors.

SVM-Fortran has the concept of data sharing which distinguishes between two different sharing types: shared data is shared between all processors in a processor set and all derived processor sets, and private data is accessible only by one processor.

Specifying a common block as shared means that this common block is shared globally, i.e. accessible by any processor in any processor set executing in the scope of that common block. Specifying a variable as shared means that this variable is shared in its scope between all processors in the active processor set and all derived processor sets. The declaration of shared data can be done in the declaration part of a program unit only.

In parallel constructs it is possible to create new instances of shared data, e.g. with a `PSHARED-` or `PSHARED_COPY`-option. These instances are shared only between processors within the same derived processor set. As a consequence of that, each derived processor set has its own instance. This shared data is uninitialized unless the `COPY`-variant is used. If the `COPY`-variant is used the content of the original common block/variable is copied to all new instances.

Specifying a common block or variable as private means that each processor in the active processor set accesses a different memory location. This variable/common block is only accessible to that processor. The declaration of private data can be done either in the declaration part of a program unit or with declaration options of parallel constructs.

Variables which are not declared to be members of a common block are allocated on a run-time stack. Different incarnations of a subprogram therefore access different shared variables.

By default all variables and common blocks are shared. This global default can be changed for a program unit default by a `DEFAULT`-directive with a scope of the surrounding program unit.

Restrictions:

- All variables in a common block must have the same sharing type, i.e. either shared or private.
- The use of the SAVE-statement is not allowed for shared data.
- Equivalenced variables shall have the same sharing type.
- Declaration of formal arguments in a subprogram as either shared or private is

seen as a hint to the compiler.

4.1 Default Specification

Syntax:

default-directive	**is**	`DEFAULT ( `*sharing*` )`
sharing	**is**	`SHARED`
	or	`PRIVATE`

◇

Description:

By default all variables in a program unit are shared. The user can overwrite this unit default with the *sharing*-option of the `DEFAULT`-directive.

Example:

```
        PROGRAM default
CSVM$   DEFAULT (PRIVATE)
        REAL a, b
CSVM$   PRIVATE:: b
        END
```

The variable *a* is private by the local default of the program unit, and *b* is private by explicitly specifying the sharing type for the variable via the `PRIVATE`-directive. ■

4.2 Shared and Private Variables

Syntax:

shared-directive	**is**	`SHARED [,ALIGN]::` *p-name-list*
private-directive	**is**	`PRIVATE ::` *p-name-list*
p-name	**is**	*variable-name*
	or	`/` *common-block-name* `/`
common-block-name	**is**	*identifier*

◇

Description:

These directives specify variables or common blocks as shared or private, respectively. Variables which are members of common blocks may be declared explicitly shared or private; this has to be done consistently with the common block sharing type.

These directives overwrite the global default or the default specified for a program unit. If one of the directives is used with a formal argument, this information will only be seen as a hint to the compiler.

The `ALIGN`-option specifies that all variables/common blocks declared in this directive shall be aligned on a page boundary. There is no alignment inside common blocks. Variables in common blocks are aligned on page boundaries if they are specified explicitly as aligned.

Restrictions:

- The directives may only be used in the declaration part of a program unit.
- The sharing type of a common block shall be the same in all program units which use the common block.
- If an alignment for variables in common blocks was given, this shall be done for all declarations of this common block.

Example: Scope of Variables

```
        PROGRAM scope
CSVM$   SHARED :: a,b
CSVM$   PRIVATE :: c,i

        a = MYPROC()    ! visible on all processors
        b = MYPROC()    ! visible on all processors
        c = MYPROC()    ! private c is modified only on p0
C***    point 1

CSVM$   PDO(LOOPS(i), STRATEGY(REPBLOCK), PRIVATE(b))
        DO i=1,2
           a = MYPROC() ! executed and seen on all processors
           b = MYPROC() ! new loop-private variable
```

```
           c = MYPROC() ! private variable
C***       point 2
        ENDDO

C***    point 3
        STOP
        END
```

In the declaration part, the variables *a* and *b* are specified as shared, and the variable *c* is specified as private.

Assume, the program is executed by three processors: p0-p2. Therefore the initial processor set ps0 contains p0, p1, p2. The first program region is executed only by p0 as the default execution mode is exclusive mode. Therefore, at point 1 the value of *a* and *b* is 0 (seen on all processors). `MYPROC()` is an intrinsic function and returns the global processor number. The value of *c* is 0 on processor p0 and undefined on p1 and p2.

On the parallel loop, two derived processor sets are generated: ps1, ps2. ps1 contains p0, p1, and ps1 contains p2. The iterations are executed in exclusive mode. A new private variable *b* is introduced. Therefore, at point 2 the value of *a* is either 0 or 2 (race condition). The value of *b* and *c* is 0 on processor p0, 2 on processor p2, and undefined on processor p1.

At point 3 the value of *a* is 0 or 2; the value of *b* is 0. Variable *c* contains 0 on processor p0, 2 on processor p2, and is undefined on processor p1. ■

Example: Using partially-shared variables

```
        PROGRAM partially_shared
        COMMON /tmp/x   ! shared by default
CSVM$ PRIVATE:: i

        x = MYPROC()    ! visible on all processors
C***    point 1
CSVM$   PDO(LOOPS(i), STRATEGY(REPBLOCK), PSHARED(/tmp/))
        DO i=1,2
           x = MYPROC() ! pshared
C***       point 2
        ENDDO
        STOP
        END
```

Assume, the program is executed by three processors: p0-p2. Therefore the initial processor set ps0 contains p0, p1, p2. The first program region is executed only by p0 as the default execution mode is exclusive mode.

At point 1 *x* contains the value 0.

On the parallel loop, two derived processor sets are generated: ps1, ps2. ps1 contains p0,p1, and ps2 contains p2. The iterations are executed in exclusive mode. A new partially shared common block *tmp* is introduced for both processor sets. At point 2 the value of *x* is 0 on p0 and p1; on p2 the value is 2. ■

Syntax:

copy-directive **is** `COPY ::` *p-name-list* ◇

Description:

Executing the `COPY`-directive means that all processors of the active processor set which are suspended, i.e. all but the master processor, update their private variable(s) or common block(s) specified in the *p-name-list* by the value(s) of the corresponding variable(s) or common block(s) of the master processor.

Restrictions:

- This directive shall be used only in exclusive regions.
- All items specified in the *p-name-list* shall be private.

Example: Use of the COPY-directive

```
        PROGRAM copy
CSVM$   PRIVATE:: x,y

CSVM$   EXCLUSIVE_REGION
C***    read values
        READ(*,*) x,y
C***    distribute copies
CSVM$   COPY:: x,y
CSVM$   EXCLUSIVE_REGION_END
        STOP
        END
```

The master processor reads (in exclusive mode) the two variables *x* and *y*. With the `COPY`-directive the values of the two variables is propagated to the private copies on all processors of the processor set. ■

5 Parallelism

Initially, the execution of an SVM-Fortran program starts with the main program, either in exclusive or replicated mode whichever applies to that region. Parallelism has to be specified explicitly. If a processor reaches the start of a parallel construct, the following happens:

1. If the parallel program region is inside an exclusive region, the other (suspended) processors in the active processor set get signaled that there is work to be done.
2. If the parallel program region needs a barrier synchronization, it is done at that point.
3. Determine the processors that share the work on the parallel program region (e.g. specified with the PROCESSORS-option in a parallel loop), and determine the work that has to be distributed to those processors (e.g. the number of iterations in a parallel loop).
4. Create derived processors sets, and make the derived processor sets active. From this time the derived processor sets work independent of each other.
5. Each derived processor set executes the code assigned to it.
6. At the end of the parallel construct, delete the new active processor set and continue with the old active processor set.
7. If the parallel program region needs a barrier synchronization, it is done at that point.
8. The old active processor set continues execution. If it is necessary (i.e. the active processor set is in an exclusive region), all processors of the active processor set other than the master processor are suspended.

All constructs introduced in this chapter may be nested. It is not allowed to jump into a program region or to jump out of a program region.

5.1 Parallel Loops

Syntax:

parloop-directive	**is**	`PDO` [`(` *parloop-option-list* `)`]
parloop-option	**is**	`LOOPS` `(` *loop-var-list* `)`
	or	`PRIVATE` `(` *p-name-list* `)`
	or	`PSHARED` `(` *p-name-list* `)`
	or	`PSHARED_COPY` `(` *p-name-list* `)`
	or	`REDUCTION` `(` *variable-name-list* `)`
	or	`NOBARRIER`
	or	`PROCESSORS` `(` *dist-target* `)`
	or	`STRATEGY` `(` *strat-spec* `)`
strat-spec	**is**	*loop-level-scheduling-list*
	or	*predefined-scheduling*
	or	*semi-dynamic-scheduling*
loop-level-scheduling	**is**	*direct-scheduling*
	or	*dynamic-scheduling*
direct-scheduling	**is**	*
	or	`BLOCK`
	or	`REPBLOCK`
	or	`CYCLIC` [`(` *int-expr* `)`]
	or	`REPCYCLIC` [`(` *int-expr* `)`]
	or	`ALIGNED` `(` *array-name* `(` *int-expr* `)` `)`
dynamic-scheduling	**is**	`SELF`
	or	`CHUNK` `(` *int-expr* `)`
	or	`GUIDED`
	or	`BLOCK_GUIDED` `(` *int-expr* `)`
	or	`FACTORING`
	or	`BLOCK_FACTORING` `(` *int-expr* `)`
	or	`GRAB_ALIGNED` `(` *array-name* `(` *int-expr* `)` *(int-expr* `)` `)`
predefined-scheduling	**is**	`ON_HOME` `(` *template-name* `(` *templ-spec-list* `)` `)`
semi-dynamic-scheduling	**is**	`DISTRIBUTE_ONCE` `(` *template-name* `(` *templ-spec-list* `),` `(` *loop-level-scheduling* `)` `)`
dist-target	**is**	*processors-name* [`(` *proc-range-list* `)`]
proc-range	**is**	*
	or	*int-expr*
	or	[*lower-bound*]`:`[*upper-bound*]
templ-spec	**is**	*
	or	*int-expr*
loop-var	**is**	*variable-name*
template-name	**is**	*identifier*
array-name	**is**	*identifier*
lower-bound	**is**	*int-expr*
upper-bound	**is**	*int-expr*

◇

Description:

The PDO directive is placed immediately in front of a single loop or a perfectly nested set of loops. The loop nest specified by the loop variables in the option LOOPS should run in parallel. Parallel loops of SVM-Fortran are different from independent loops of HPF as parallel loops in SVM-Fortran can include synchronization and thus iterations can depend on other iterations. If the LOOPS-option is missing, the following loop will be used.

Arriving at the parallel loop the active processor set is split up to form new derived processor sets working on independent loop iterations. How the active processor set is split and how the iterations are distributed to the processors depends on the PROCESSORS-option (if given) and on the scheduling strategy. The iterations are executed in exclusive mode or replicated mode depending on the execution mode which was active at the beginning of the parallel loop. Before and after the parallel loop a barrier synchronizes all processors in the active processor set. NOBARRIER omits the barrier synchronization at the beginning and at the end of the outermost parallel loop.

It is possible to introduce variables or common blocks as *private* or *shared* within the loop by the use of PRIVATE, PSHARED, and PSHARED_COPY. Introduced shared variables are only shared between the processors of the derived processor sets. Variables in a PRIVATE-, PSHARED-, or PSHARED_COPY declaration option inherit all attributes (e.g. type, dimension) from their corresponding declarations in the declaration part of the program unit, or the default attributes if not declared explicitly. On the other side, the memory association given in an EQUIVALENCE-statement is lost for those private or shared variables.

With the REDUCTION-option reduction variables can be given for which the compiler automatically generates appropriate code. Supported reduction operations are +,-, *, MAX, and MIN. The reduction operations shall have the form *var-expr = var-expr reduction-operator expr*. *var-expr* may be an array-expression, e.g. *a(i)*. The same reduction variable may be used in more than one reduction of the same type.

The PROCESSORS-option may be used together with the *loop-level-scheduling* rules of the STRATEGY-option to schedule one- or multi-dimensional loops. The meaning of the *proc-range*-arguments is as follow:

- '*' means the replication in this dimension of the processors arrangement.
- int-expr specifies one index in that dimension.
- *lower-bound:upper-bound* marks a region in this dimension. A single colon marks the whole dimension.

If the PROCESSORS-option is missing, the active processor set is used implicitly as a one-dimensional processor arrangment. The processors specified either implicitly or with the PROCESSORS-argument are called assigned processors.

The STRATEGY-option specifies the loop scheduling technique. All iterations of the loop(s) are distributed to processors of the active processor set or a subset of it if the

PROCESSORS-option is used. There are three different scheduling types available which will be described in detail later:

1. loop-level scheduling
2. predefined scheduling
3. semi-dynamic scheduling

Restrictions:

- Loop variables shall be private variables.
- It is not allowed to declare private data as pshared.
- A reduction variable shall be used only in the reduction operation.
- All processors specified in the PROCESSORS option shall be in the active processor set.
- With *predefined scheduling* and *semi-dynamic scheduling* all processors to which work is assigned shall participate in the execution of the loop.
- The *processors-name* in the PROCESSORS option shall be no dummy argument of the subprogram.
- DISTRIBUTE_ONCE shall be used together with the PROCESSORS-option only.
- With loop-level scheduling the length of the scheduling list must be equal to the number of parallel loops.
- With semi-dynamic scheduling the number of non-'$*$' entries in the *loop-level-scheduling-list* has to be equal to the number of regions in the *proc-range-list* or to the rank of the processor arrangement if the list is omitted.

loop-level scheduling:

Loop-level scheduling distinguishes between direct and dynamic scheduling strategies. Direct strategies are:

- '$*$' marks a loop as sequential. The loop is executed by all processors assigned to that loop.
- The `BLOCK` strategy chooses a block size such that the loop iteration space is covered by the processors assigned to that loop. The block distribution works as in HPF.
- The `REPBLOCK` strategy distributes all iterations of the loop to the processors assigned to that loop in a block fashion. Any assigned processor has at most one iteration more than any other assigned processor. If more assigned processors than iterations are available, the assigned processors are distributed in blocks to iterations. The number of processors assigned to iterations differ at most by one.
- A cyclic distribution with a given block size is given with the `CYCLIC`-option.
- The `REPCYCLIC` strategy works as the `CYCLIC`-strategy, except that more than one processor may work on an iteration if there are more processors in the active processor set than iterations are available.
- With the `ALIGNED` strategy, iterations are executed on that processor who owns the page marked with the expression *array-name(int-expr)*. The evaluation of the owner is done synchronously on all processors of the active processor set with respect to the entry of the loop. If there is an iteration not assigned to any processor because no processor of the active processor set owns the page corresponding to that iteration, this iteration is done by some processor of the assigned processors.

Dynamic strategies are:

- `SELF` chooses self scheduling.
- `CHUNK` chooses chunk scheduling where the chunk size is determined by the *int-expr*.
- `GUIDED` chooses guided self scheduling.
- `BLOCK_GUIDED` is the same as `GUIDED` but the number of iterations found in each step is rounded up to a multiple of the block size given with *int-expr*.
- `FACTORING` chooses the factoring algorithm.
- `BLOCK_FACTORING` is the same as `FACTORING` but the number of iterations found in each step is rounded up to a multiple of the block size given with *int-expr*.

- With the GRAB_ALIGNED strategy, iterations are executed on that processor who owns the page marked with the expression *array-name(int-expr)*. The evaluation of the owner is done synchronously on all processors of the active processor set with respect to the entry of the loop. If there is an iteration not assigned to any processor because no processor of the active processor set owns the page corresponding to that iteration this iteration is done by some processor of the assigned processors. If a processor has no more work to do, it tries to *steel* pages (i.e. iterations) from other processors. The number of pages is determined by the second parameter *int-expr* to the GRAB_ALIGNED option.

Example: Direct scheduling without barrier synchronization

```
CSVM$   PDO(LOOPS(i,j),
CSVM$+       STRATEGY(BLOCK,BLOCK),
CSVM$+       NOBARRIER,
CSVM$+       PRIVATE(tmp)
CSVM$+      )
        DO i=1,n
           DO j=1,m
              tmp = ...
              ...
           ENDDO
        ENDDO
```

Both loops are scheduled in blocks to all processors of the active processor set. There is no barrier synchronization at the beginning and at the end of the loops. The variable *tmp* is private to that loop. ■

Example: Using REPBLOCK

```
CSVM$   PDO(STRATEGY(REPBLOCK))
        DO i=1,2
           ...
CSVM$      PDO
           DO j=1,m
              tmp(j) = ...
              ...
           ENDDO
           ...
        ENDDO
```

As there are only 2 iterations of the outer loop the REPBLOCK-strategy is used. If for example 50 processors reach this loop two new processor sets are generated each consisting of 25 processors. The second parallel loop is then executed by 25 processors for each iteration of the outer loop. If the BLOCK-strategy would have been used, each derived processor set would consist of one processor only. ■

Example: Using the PROCESSORS-option with replication

```
CSVM$   PROCESSORS:: P(2,3)
CSVM$   PDO(LOOPS(i),
CSVM$+      STRATEGY(BLOCK),
CSVM$+      PROCESSORS(P(*,:))
CSVM$+     )
        DO i=1,N
           ...
        ENDDO
```

The loop is replicated on all processors in the first dimension and distributed to all processors in the second dimension of processor arrangement *P*. Assume, *N*=6. Then processors *P(1,1)*, *P(2,1)* work on iterations 1 and 2, *P(1,2)* and *P(2,2)* work on iterations 3 and 4, and *P(1,3)* and *P(2,3)* work on iterations 5 and 6. ■

Example: Loop-level scheduling for nested parallel loops

```
        PROGRAM using_myprocind
CSVM$   PROCESSORS:: P(2,3)
CSVM$   PDO (LOOPS(i), PROCESSORS(P(:,*)),STRATEGY(BLOCK))
        DO i=1,100
                 ...
CSVM$      PDO (LOOPS(j),
CSVM$+          PROCESSORS(P(MYPROCIND(P,1,IERR)),:),
CSVM$+          STRATEGY(BLOCK))
           DO i=1,100
              ...
           ENDDO
              ...
        ENDDO
        END
```

Each iteration of the outer loop is replicated onto 3 processors. The iterations of the inner loop are distributed to these 3 processors. ■

Example: Aligned scheduling

```
        DIMENSION a(N)
CSVM$   PDO(STRATEGY(ALIGNED(a(i))))
        DO i=1,N
           a(i) = ...
        ENDDO
```

The loop is scheduled according to the page ownership of the locations addressed with *a(i)* for iteration *i*. If none of the processors in the active processor set is the owner of page *a(j)* for some *j*, an arbitrary processor of the active processor set will do iteration *j*. ■

predefined scheduling:

With predefined scheduling the scheduling of the loop iterations is done according to a template's distribution. A template is an abstract index space. The elements can be distributed onto the abstract processors (see Section 7).

The ON_HOME clause assigns iterations to processors owning the template elements. The subscripts in the *templ-spec-list* can have the following form:

- loop variable of the parallel loop

  The iterations of this loop are scheduled according to the distribution of that template dimension. Each loop variable shall be used at most once in *templ-spec-list*. If a loop variable is not used in the *templ-spec-list* the corresponding loop will be executed sequentially.

- *int-expr* independent of the loop variables

  The expression determines a single index of the template in that dimension.

- '$*$' specifying replication

  In the distribution directive the template dimension was either distributed to an entire dimension of a processor arrangement or explicitly to a range of indices. The '$*$' replicates the iterations to all processors with an arbitrary index in that processor arrangement dimension or, in the second case, an index in the specified range.

Example: Predefined scheduling

```
CSVM$   PROCESSORS::  P(4,4)
CSVM$   TEMPLATE:: T(10,10)
CSVM$   DISTRIBUTE (BLOCK,BLOCK) ONTO P:: T

        ...
CSVM$   PDO(LOOPS(I,J),STRATEGY(ON_HOME(T(I,J))))
        DO I=1,10
           DO J=1,10
              ...
           ENDDO
        ENDDO

CSVM$   PDO(Loops(I,J),STRATEGY(ON_HOME(T(I,J))))
        DO I=1,10
           DO J=1,10
              ...
           ENDDO
        ENDDO
```

Both loops have the same scheduling strategy. The strategy can be changed globally by specifying another distribution for the template. The scheduling of the loops can be adapted dynamically by redistributing the template according to the workload of the processors.

■

Example: Predefined scheduling

```
CSVM$   PROCESSORS:: P(2,3)
CSVM$   TEMPLATE:: TEMPL(N, M)
CSVM$   DISTRIBUTE (BLOCK, BLOCK) ONTO P:: TEMPL
CSVM$   PDO(STRATEGY(ON_HOME(templ(*,i))))
        DO i=1,N
           ...
        ENDDO
```

Predefined loop scheduling is used. The loop is replicated in the first dimension of the template `templ` and distributed according to the second dimension. ■

semi-dynamic scheduling:

`DISTRIBUTE_ONCE` stores the work distribution in a template in order to apply the same distribution to another parallel loop nest of the same size, or for another execution of the same loop.

If the template has no defined distribution when the loop is executed, the loop is scheduled according to the strategy given in *loop-level-scheduling-list* and the schedule determines the distribution of the template. Otherwise, i.e. the template has a defined distribution, this distribution is used for scheduling.

The subscripts of the *templ-spec-list* can have the following form:

- loop variable of the parallel loop

  The schedule for the iterations of this loop determines the distribution of the template dimension. Each loop variable shall be used at most once in *templ-spec-list*.

- '*'

  This template dimension is sequentialized (see example below).

Example: Semi-dynamic scheduling

```
CSVM$   TEMPLATE:: TEMPL(N, M)
        DO i=1,maxiter
CSVM$     PDO(STRATEGY(DISTRIBUTE_ONCE(TEMPL(i, *),(SELF))))
          DO i=1,N
             ...
          ENDDO
        ENDDO
```

Semi-dynamic scheduling is used in this example. If the distribution of the template *templ* is defined already, this distribution is used for loop scheduling. Otherwise, the template's distribution is *undefined*, self scheduling is used and the distribution of the loop iterations to the processors is stored in the template. ■

5.2 Parallel Sections

Syntax:

psection-directive	**is**	`PSECTION` [`(` *psection-option-list* `)`]
		psection-list
		`PSECTION_END`
psection	**is**	`SECTION` [*section-option-list*]
		statement-list
psection-option	**is**	`PRIVATE` `(` *p-name-list* `)`
	or	`PSHARED` `(` *p-name-list* `)`
	or	`PSHARED_COPY` `(` *p-name-list* `)`
	or	`NOBARRIER`
section-option	**is**	`ON` *processor-region*
processor-region	**is**	*processors-name* [`(` *proc-region-list* `)`]
proc-region	**is**	[*lower-bound*]`:`[*upper-bound*]
	or	*int-expr*
processors-name	**is**	*identifier*

◇

Description:

Several code sections should be run in parallel by the active processor set. The individual sections are executed by derived processor sets in the execution mode which applies to the beginning of the parallel region. Before and after the parallel section a barrier synchronizes all processors of the active processor set. The option `NOBARRIER` omits the barrier synchronization at the beginning and at the end of the parallel section.

If no explicit scheduling strategy is given (i.e. with the `ON`-option), the default strategy is to distribute sections in blocks to processors such that any processor has at most one section more than any other processor. If there are more processors than sections, processors are assigned to sections such that on every section works at most one processor more than on any other section.

Variables in a `PRIVATE`-, `PSHARED`-, or `PSHARED_COPY`-declaration option inherit all attributes (e.g. type, dimension) from their corresponding declarations in the declaration part of the program unit, or the default attributes if not declared explicitly.

Restrictions:

- If the `ON`-option is used on a section of a parallel section, all sections shall use the `ON`-option.
- Processor regions specified by `ON`-options of all sections shall result in disjoint sections of the active processor set.
- It is not allowed to declare private data as pshared.

Example: Two parallel sections

```
        PROGRAM section
CSVM$   PSECTION(NOBARRIER)
CSVM$   SECTION
        DO i=1,n
           ix(i) = i
        ENDDO
CSVM$   SECTION
        DO i=1,m
           iy(i) = i
        ENDDO
CSVM$   PSECTION_END
        END
```

On entering the parallel section two derived processor sets work on the two sections.

■

Example: Explicit scheduling using the ON-option

```
        PROGRAM section_with_on
CSVM$   PROCESSORS:: p1(8,2), p2(4)
CSVM$   PSECTION
CSVM$   SECTION ON p1(2:7,:)
        DO i=1,n
           ix(i) = i
        ENDDO
CSVM$   SECTION ON p2
        DO i=1,m
           iy(i) = i
        ENDDO
CSVM$   PSECTION_END
        END
```

The first section is done by a derived processor set consisting of a subset of processor arrangement *p1*. The subset is the region of the processor set *p1* marked with 2-7 in the first dimension and all processors of the second dimension. The second section is executed by a derived processor set consisting of all processors of the processor arrangement *p2*. ■

5.3 Replicated and Exclusive Regions

Syntax:

repregion-directive	**is**	REPLICATED_REGION[(*region-option-list*)] *statement-list* REPLICATED_REGION_END
exclregion-directive	**is**	EXCLUSIVE_REGION[(*region-option-list*)] *statement-list* EXCLUSIVE_REGION_END
region-option	**is**	PRIVATE (*p-name-list*)
	or	NOBARRIER

◇

Description:

A code segment is executed in replicated mode (REPLICATED_REGION) or exclusive mode (EXCLUSIVE_REGION) by the active processor set. Before and after a replicated and exclusive region a barrier synchronizes the processors in the active processor set. NOBARRIER omits the barrier synchronization at the beginning and at the end of the region.

Variables in a PRIVATE-, PSHARED-, or PSHARED_COPY-option inherit all attributes (e.g. type, dimension) from their corresponding declarations in the declaration part of the program unit, or the default attributes if not declared explicitly.

The control flow in a replicated region is independent of the control flow of other processors executing in the replicated region. See Section 3 for a detailled description of *replicated mode* and *exclusive mode*.

Restrictions:

- It is not allowed to declare private data as pshared.

Example: Nesting of an exclusive and replicated region

```
        PROGRAM region
CSVM$   REPLICATED_REGION
        ...
CSVM$      EXCLUSIVE_REGION
           ...
CSVM$         PDO
              DO i=1,n
                 ix(i) = i
              ENDDO
CSVM$      EXCLUSIVE_REGION_END
CSVM$   REPLICATED_REGION_END
        END
```

In the example given, the replicated region is executed by all processors in the active processor set. Starting with the exclusive region, one processor of the active processor set executes the

code exclusively. The iterations of the parallel loop can be executed in parallel by processors of derived processor sets. The execution mode of each iteration is the exclusive mode. ■

Example: Replicated iterations

```
        PROGRAM region
        PARAMETER(N=100000, M=40)
CSVM$   PROCESSORS:: p(2,NUMPROC()/2)
CSVM$   TEMPLATE:: templ(M,N)
CSVM$   DISTRIBUTE (BLOCK,BLOCK) ONTO p:: templ

CSVM$   REPLICATED_REGION
CSVM$   PDO(STRATEGY(ON_HOME(templ(*,i))))
        DO i=1,N
           ...
        ENDDO
CSVM$   REPLICATED_REGION_END
        END
```

The loop is replicated in the first dimension of the template *templ* and distributed according to the second dimension. ■

Syntax:

coordinated-call-directive **is** COORDINATED_CALL [(*subprog-name-list*)]
subprog-name **is** *identifier* ◇

Description:

The subprogram calls as given by the *subprog-name-list* (subroutine or function names) in the following statement are executed by all processors of the active processor set within one processor set. If no subprograms are given all calls in the following statement will be handled as coordinated calls. This overwrites the default for replicated regions that every processor executes a subprogram call in a derived processor set.

Restrictions:

- This directive can only be used in a replicated region.

Example: Calling a subroutine with a coordinated call

```
        PROGRAM subtest
        DIMENSION a(100), component_sum(100)
CSVM$ PRIVATE:: i
CSVM$ PROCESSORS:: P(4)

CSVM$   REPLICATED_REGION
           i = myprocset() * 25 + 1
C***       executed by every processor independently
C***       in derived processor sets
           CALL sub(a(i),b(i),component_sum(i),25)

C***       executed by all processors with a coordinated call
CSVM$      COORDINATED_CALL
           CALL sub(a,b,sum,100)
CSVM$   REPLICATED_REGION_END
        END

        SUBROUTINE sub(a,b,res,n)
        DIMENSION a(n),b(n),res(n)
CSVM$   PRIVATE:: i
CSVM$   PDO(REDUCTION(res))
        DO i=1,n
          res = res + a(i) + b(i)
        ENDDO
        RETURN
        END
```

Assume, the program is executed by 4 processors. The replicated region in the main program is executed by all processors in the active processor set. The first call to subroutine *sub* is executed by every processor of the active processor set independently. Every derived

processor set consists of one processor. In the subroutine *sub*, the parallel loop is therefore executed `NUMPROC()` times, each loop with 25 iterations executed by one processor (as each derived processor set consists of exactly one processor). The second call to *sub* is executed by all processors of the active processor set, the active processor set remains. The parallel loop in *sub* is executed once, all processors share the work on n=100 iterations. ■

6 Synchronization

SVM-Fortran provides three different types of synchronization. Barrier synchronization is used to synchronize all processors in a processor set. Critical sections ensure that at most one processor executes some piece of code. Lock synchronization is used to ensure that at most one processor accesses some piece of data.

Synchronization constructs shall not be nested. It is not allowed to jump into or to jump out of a synchronization construct.

6.1 Barrier Synchronization

Syntax:

barrier-directive	**is**	BARRIER [*barrier-option*]
	or	BARRIER_CHECKIN [*barrier-option*]
		statement-list
		BARRIER_CHECKOUT
barrier-option	**is**	ON *processors-name-list*

◇

Description:

A barrier synchronizes all processors in the active processor set. A processor continues execution only if all other processors in the active processor set have reached this point. Between the BARRIER_CHECKIN and the BARRIER_CHECKOUT directive additional code is allowed which is executed according to the execution mode which applies to this code. Processors are blocked at the BARRIER_CHECKOUT point until all processors of the active processor set have reached the BARRIER_CHECKIN point. In an exclusive region suspended processors are reactivated for synchronization.

With the ON-option it is possible to synchronize different processor arrangements.

Restrictions:

- BARRIER_CHECKIN and BARRIER_CHECKOUT options shall be paired and cannot be nested.
- Inside the code between a BARRIER_CHECKIN and BARRIER_CHECKOUT no other barrier synchronization constructs are allowed either explicitly or implicitly (e.g. parallel loops).
- If the ON option is used, the processor arrangements shall not share any common processor with respect to the global numbering as given by MYPROC.

Example: Separating two different phases of a computation

```
        PROGRAM two_phases
CSVM$   REPLICATED_REGION
        DO i=lower_bound1, upper_bound1
           a(i) = ...
        ENDDO
CSVM$   BARRIER
        DO i=lower_bound2, upper_bound2
           ... = a(i)
        ENDDO
CSVM$   REPLICATED_REGION_END
        END
```

The two phases of computation are separated by a barrier synchronization. ■

Example: Using BARRIER_CHECKIN/CHECKOUT

```
        PROGRAM checkin_checkout
CSVM$   REPLICATED_REGION
        DO i=lower_bound1, upper_bound1
           a(i) = ...
        ENDDO
CSVM$   BARRIER_CHECKIN
        CALL independent_work()
CSVM$   BARRIER_CHECKOUT
        DO i=lower_bound2, upper_bound2
           ... = a(i)
        ENDDO
CSVM$   REPLICATED_REGION_END
        END
```

■

Example: Using mutual synchronization

```
        PROGRAM mutual_sync
CSVM$   PROCESSORS:: p1(numprocset()/2), p2(numprocset()/2)
CSVM$   PSECTION
CSVM$   SECTION ON p1
        DO i=1,n
           a(i) = ...
        ENDDO
CSVM$   BARRIER ON p1,p2
        DO i=1,n
           ... = b(i)
        ENDDO

CSVM$   SECTION ON p2
        DO i=1,n
           b(i) = ...
        ENDDO
CSVM$   BARRIER ON p1,p2
        DO i=1,n
           ... = a(i)
        ENDDO
CSVM$   PSECTION_END
        END
```

This example shows how to use mutual synchronization. Two processor arrangements *p1* and *p2* are defined. The first parallel section is executed by processors in processor arrangement *p1* while the second parallel section is executed by processors in *p2*. To avoid data conflicts, every section contains a synchronization of both processor arrangements. ■

6.2 Critical Section

Syntax:

critical-section-directive **is** `CRITICAL_SECTION`
statement-list
`CRITICAL_SECTION_END` ◇

Description:

No more than one processor can execute the code given with *statement-list*. If a processor *p* reaches the start of the critical section while another processor is inside the critical section, the processor *p* blocks at the beginning of the critical section.

Restrictions:

- Critical sections shall not be nested.

Example: Using a critical section

```
        PROGRAM critical
CSVM$   REPLICATED_REGION
        ...
CSVM$   CRITICAL_SECTION
        count = count + 1
CSVM$   CRITICAL_SECTION_END
        ...
CSVM$   REPLICATED_REGION_END
        END
```

■

6.3 Lock Synchronization

Syntax:

lock-directive	**is**	`LOCK` *memory-region-list*
		statement-list
		`LOCK_END`
memory-region	**is**	*variable-name* [(*region-spec-list*)]
region-spec	**is**	*lower-bound* : *upper-bound*
	or	*int-expr*

◇

Description:

A processor continues execution only if it can acquire globally exclusive access to the memory regions given by the *memory-region-list*. The locked memory region can be released with the directive `LOCK_END` allowing other processors to acquire it.

Restrictions:

- `LOCK` and `LOCK_END` directives have to be paired and cannot be nested.
- The programmer must be aware of the fact that certain systems impose restrictions on the granularity of memory locks (e.g. a whole page will be locked).

Example: Lock a memory region

```
        PROGRAM lock
        DIMENSION x(1024)
CSVM$   PDO(LOOPS(i))
        DO i=1,n,1024
           ...
CSVM$      LOCK (x(i))
           x(i) = x(i) + ...
CSVM$      LOCK_END
           ...
        ENDDO
        END
```

■

6.4 Update Operation

Syntax:

atomic-update-directive **is** `ATOMIC_UPDATE`
assignment-statement ◇

Description:

The directive ensures an atomic update of the left side of the assignment statement. This directive is a user-friendly notation for locking and unlocking the referenced location on the left side of the assignment.

Example: Atomic Update

```
        PROGRAM atomic_update
        DIMENSION x(1000)
        ...
CSVM$   ATOMIC_UPDATE
        x(i) = x(i) + 1
        ...
        END
```

■

7 Work Distribution

In SVM-Fortran the mapping of work to processors can be defined by work distribution directives. The language features to denote a global work distribution are based on the template concept of High Performance Fortran (HPF). In contrast to other languages, there is no way to specify data distributions explicitly. The distribution of data results from the distribution of work due to dynamic data movement in the multiprocessor.

7.1 Processor Arrangements

Syntax:

processors-directive	**is**	PROCESSORS:: *processors-specification*
processors-specification	**is**	*processors-decl-list*
	or	*processors-decl* RESHAPE:: *processors-decl-list*
processors-decl	**is**	*processors-name* (*explicit-shape-spec-list*)
explicit-shape-spec	**is**	[*lower-bound* :] *upper-bound*

◇

Description:

The PROCESSORS directive declares one or more rectilinear processor arrangements, specifying for each one its name and its shape. Processor arrangements are used to identify abstract processors in work distribution annotations.

They can be of fixed size or dependent on the execution context, e.g. the intrinsic functions NUMPROC() and NUMPROCSET() can be used in the declaration to specify arrangements depending on the number of available processors.

The RESHAPE attribute adopted from Vienna Fortran provides a mechanism to allow the same processor set to be viewed as having different rectilinear geometries, possibly of differing rank. The mapping of processor arrangement elements is done in column major order.

Processor arrangements are mapped to the active processor set according to the sequence of their declaration. Each processor arrangement is mapped by this rule onto a distinct set of processors.

Unlike HPF, SVM-Fortran processor arrangements can be arguments to subprograms. Dummy processor arrangements are not mapped onto the active processor set but identify the processors denoted by the actual argument.

Restrictions:

- The PROCESSORS directive shall appear only in the specification part of a program unit.
- Every dimension of a processor arrangement must have nonzero extent; therefore a processor arrangement cannot be empty.
- A processor arrangement is an object global to the processors in the active processor set. Different processors in the active processor set shall not specify a different shape for the same processor arrangement.
- It is not allowed to assign work to a dummy processor arrangement. Work can be assigned to processors in the active processor set. A dummy processor arrangement may only be used in a BARRIER-directive or in explicit message passing.
- The shape of a dummy processor arrangement and the shape of the corresponding actual argument must match. A different view on the actual processor

arrangement can be provided explicitly through a RESHAPE declaration.

Example: Processor mapping

```
      PROGRAM PROC
CSVM$ PROCESSORS:: P(NUMPROC())

         ...
      k = NUMPROC() / 2
CSVM$ PSECTION
CSVM$ SECTION ON P(1:k)
        call F()
CSVM$ SECTION ON P(k+1:NUMPROC())
        call G()
CSVM$ PSECTION_END
       ...
      END
```

The size of processor arrangement *P* is determined according to the number of processors assigned to the application. The parallel sections are then mapped via *P* onto the processors.

```
      SUBROUTINE F()
CSVM$ PROCESSORS:: P1(NUMPROCSET())
           ...
      END


      SUBROUTINE G()
CSVM$ PROCESSORS:: P1(2,2)
CSVM$ PROCESSORS:: P2(NUMPROCSET()-4)

           ...
      END
```

The processor arrangements in the subroutines are mapped to the active processor set executing the individual sections. NUMPROCSET() returns the number of processors in the active processor set. In subroutine *G* the following processor mapping occurs:

```
P1(1,1) --> P(k+1)
P1(2,1) --> P(k+2)
P1(1,2) --> P(k+3)
P1(2,2) --> P(k+4)
P2(1)   --> P(k+5)
        ...
```

■

Example: Reshaping

```
        PROGRAM RESH
CSVM$   PROCESSORS:: P(8,8) RESHAPE:: Q(64), R(4,4,4)
```

Processor arrangement *Q* provides a one-dimensional view for *P*. Processor arrangement *R* is a three-dimensional view of *P*. The use of RESHAPE in combination with the DISTRIBUTE directive is shown in example "RESHAPE and DISTRIBUTE" on page 52.

■

Example: Synchronization of processor sets

```
CSVM$    PROCESSORS:: P1(4,4),P2(4,4)
         COMMON /XX/ A
         INTEGER A(100)
CSVM$    SHARED:: XX

CSVM$    PARALLEL_SECTION
CSVM$    SECTION  ON P1
            CALL F(...,P2)
CSVM$    SECTION  ON P2
            CALL G(...,P1)
CSVM$    PARALLEL_SECTION_END

         END



         SUBROUTINE F(...,Q)
CSVM$    PROCESSORS:: P(4,4),Q(4,4)
         COMMON /XX/ A
         INTEGER A(100)
CSVM$    SHARED:: XX
         ...

CSVM$    BARRIER  ON (P,Q)
         ...
         END


         SUBROUTINE G(...,R)
CSVM$    PROCESSORS:: P(4,4),R(4,4)
         COMMON /XX/ A
         INTEGER A(100)
CSVM$    SHARED:: XX
         ...

CSVM$    BARRIER ON (P,R)
         ...
         END
```

Processor sets *P1* and *P2* execute a different section of the parallel section. *P1* calls *F* and passes *P2* through the subroutine interface. The dummy processor arrangements *P* and *R* are not mapped onto the active processor set. Inside *F*, *P1* is mapped to *P* and *P2* is mapped to *Q*. Thus, *P1* and *P2* are synchronized at the barrier. It is important that *P2* must encounter a barrier, too. Otherwise, *P1* will hang forever. *P2* calls *G* and is mapped to *P* in *G*. Here it encounters the barrier that synchronizes *P* with *R* which was mapped to the original *P1* on entry to the subroutine *G*.

■

7.2 Template Declaration

Syntax:

template-directive	**is**	`TEMPLATE [,STATIC]::` *template-decl-list*
template-decl	**is**	*template-name* (*explicit-shape-spec-list*)
	or	*template-name* (*deferred-shape-spec-list*)
deferred-shape-spec	**is**	:

◇

Description:

The `TEMPLATE` directive declares one or more templates, specifying for each the name and its shape. A template is a rectilinear index domain used for global work distribution specification. Templates are distributed to the processors in the active processor set and determine the scheduling of parallel loops via predefined scheduling.

Templates can be either of fixed size, automatic, or creatable. A creatable template is declared with a *deferred-shape-spec-list* in the template directive. The rank of the template is equal to the number of colons. Afterwards the template can be explicitly created with the `CREATE` directive at run-time. The lower and upper bounds of each dimension are those specified in the `CREATE` directive.

The `STATIC` attribute asserts that a template has a static distribution, i.e. it cannot be redistributed dynamically. Thus, the compiler can do optimizations based on the specified distribution.

The initial distribution of a template is undefined unless the template is distributed by a `DISTRIBUTE` directive. Templates with an undefined distribution cannot be used in predefined scheduling. The distribution of a template can be set undefined with the `UNDEF`-directive.

Templates can be dummy arguments to a subprogram. This enables the programmer to use the same work distribution scheme across subprogram boundaries. If a template is passed as an argument and redistributed in a subroutine, it keeps the new distribution on exit from the subroutine. This enables the programmer to determine a distribution scheme in a subroutine and to pass it to the caller.

All processors in the active processor set when execution of the subroutine with the template's declaration is started, shall execute the following operations on templates: create the template, destroy the template, change the template's distribution.

Restrictions:

- The `TEMPLATE` directive shall appear in the specification part of a program unit only.
- A template is an object global to the processors in the active processor set. The processors shall not specify a different shape for the same template.
- Templates with a static distribution cannot be redistributed by the `REDISTRIBUTE` directive or the `DISTRIBUTE_ONCE` parallel loop option.

The template's distribution can also not become undefined.

- A template that is an argument to a subprogram, shall match the declaration of the dummy template. If a template with a different shape is needed, a new template shall be declared and it shall be distributed with respect to the dummy template.

Example: Template declarations

```
        PARAMETER ( M = 30 )
CSVM$   TEMPLATE:: T(M,M)

           ...
        CALL G(..., T,M, ...)
           ...

        END


        SUBROUTINE G(....,T3,N,...)
CSVM$   TEMPLATE:: T1(10,10)
CSVM$   TEMPLATE, STATIC :: T2(N,N)
CSVM$   TEMPLATE:: T3(N,N)
CSVM$   TEMPLATE:: T4(:,:)
```

In this example, *T1* is a fixed size template, *T2* an automatic template with a static distribution, *T3* a template dummy argument, and *T4* a creatable template.

■

7.3 Creatable Templates

Syntax:

create-directive **is** `CREATE::` *template-name* (*explicit-shape-spec-list*) ◇

Description:

The `CREATE` directive is an executable statement. It creates a template that must have been declared previously as a creatable template.

A `CREATE` directive has to be executed by all processors in the processor set that was active when the template was declared. The operation is enclosed in barrier synchronizations.

Example: Creation of templates

```
      PROGRAM CREATE
CSVM$ TEMPLATE:: T(:,:), T2(:)
        ...
      READ(*,*) K, M, N
CSVM$ CREATE:: T(K,M), T2(N)
        ...
```

The templates are created dynamically according to run-time parameters specifying the actual problem size to be computed. ■

Syntax:

destroy-directive **is** `DESTROY::` *template-name-list* ◇

Description:

The `DESTROY` directive is an executable statement. It destroys a template that must have previously been created by use of the `CREATE`-directive.

A `DESTROY` directive has to be executed by all processors in the processor set that was active when the template was declared. The operation is enclosed in barrier synchronizations.

Example:

```
      PROGRAM DESTROY
CSVM$ PROCESSORS:: P(5)
CSVM$ TEMPLATE:: T(:)

        ...
 5    CONTINUE
      READ(*,*) K
CSVM$ CREATE:: T(K)

CSVM$ REDISTRIBUTE (BLOCK) ONTO P :: T

CSVM$ PDO(LOOPS(I), STRATEGY(ON_HOME(T(I)))
      DO I = 1, K
        ...
      ENDDO

CSVM$ DESTROY:: T
      GOTO 5
      ...
      END
```

In this example, template *T* is used only for the small portion of the program.

■

7.4 User-defined Template Distribution

Syntax:

distribute-directive	**is**	`DISTRIBUTE` *dist-strategy* `::` *template-name-list*
redistribute-directive	**is**	`REDISTRIBUTE` *dist-strategy* `::` *template-name-list*
dist-strategy	**is**	*standard-distribution*
	or	*indirect-distribution*
	or	*linked-distribution*
standard-distribution	**is**	`(` *dist-spec-list* `)` `ONTO` *dist-target*
indirect-distribution	**is**	`(` *index-var-list* `)` `ONTO` *processor-region*
linked-distribution	**is**	`(` *index-var-list* `)` `ONTO` *template-home*
template-home	**is**	`HOME (` *template-name* `(` *template-align-spec-list* `))`
dist-spec	**is**	*dist-function*
	or	*sequentialization*
sequentialization	**is**	*
dist-function	**is**	`BLOCK`
	or	`CYCLIC [(` *int-expr* `)]`
	or	`GENERAL_BLOCK (` *int-array* `)`
template-align-spec	**is**	*int-expr*
	or	[*lower-bound*]:[*upper-bound*]
index-var	**is**	*variable-name*

◇

Description:

The `DISTRIBUTE` and `REDISTRIBUTE` directives specify a mapping of template elements to abstract processors. This distribution can then be used in predefined loop scheduling to direct loop scheduling globally. The `DISTRIBUTE` directive may appear only in the specification part of a program unit whereas the `REDISTRIBUTE` directive is an executable statement.

A `DISTRIBUTE` directive is descriptive for dummy templates. A dummy template need not have a distribution specification but when a distribution is specified the compiler can do optimizations based on this information.

Restrictions:

- A template shall be distributed onto processors in the active processor set only.
- A creatable template can only be distributed after its creation.
- All processors in the active processor set have to specify the same distribution for a template.
- The lengths of the *dist-format-list* and of the *index-var-list* have to be equal to the rank of the template.
- An *index-var-list* is a list of distinct variables.

- If a distribution is given for a dummy template it must match the distribution of the corresponding template argument.
- A `REDISTRIBUTE` directive has to be executed by all processors in the processor set that was active when the template was declared. The operation is enclosed in barrier synchronizations.
- A template may be redistributed via a `REDISTRIBUTE` directive at any time, unless it has a static distribution.

SVM-Fortran provides different distribution strategies: standard distribution, indirect distribution, and linked distribution.

standard distribution:

Each template dimension is either sequentialized or mapped to an individual processor arrangement dimension by a one-dimensional distribution function.

The processor arrangement is specified by *dist-target*. The indices in the specification are either a range, '$*$', or an integer expression.

- range

  A range specifies the processor indices to which a template dimension is distributed.

- '$*$'

  No template dimension is distributed to that processor dimension. Template elements are replicated in that processor dimension.

- integer expression

  No template dimension is distributed to that processor dimension. Template elements are only mapped to processors with the specified index.

If only the name of a processor arrangement is given, its rank has to be equal to the number of non-'$*$' *dist-specs*. Here, the l-th template dimension with a non-'$*$' specification is mapped to the l-th processor dimension.

If an index list is given in *dist-target*, the number of non-'$*$' *dist-specs* must equal the number of ranges in the processor specification. The l-th template dimension with a non-'$*$' specification is mapped to the l-th processor dimension with a range notation.

SVM-Fortran provides the following one-dimensional predefined distribution functions:

- `BLOCK`

  This specifies the block-wise distribution known from HPF.

- `CYCLIC`[*(int-expr)*]

  The template indices are mapped in blocks of length *int-expr* cyclically onto the processor indices. This is equivalent to the cyclic distribution known from HPF. The default block length is one.

- `GENERAL_BLOCK`*(int-array)*

  The general-block distribution scheme introduced by Vienna Fortran allows the programmer to balance the work load among the processors via a block distribution with variable block length. The length of the block assigned to each processor (except the last one) is stored in an integer array which is used as a parameter of the `GENERAL_BLOCK` distribution strategy. The sum of these block lengths shall not be larger than the size of the template. The remaining template elements are assigned to the last processor.

Template dimensions are sequentialized via '$*$'. Template elements with any index in this dimension are mapped to the processors determined according to the other indices.

Example: Template distribution

```
CSVM$ DISTRIBUTE (BLOCK,*,BLOCK) ONTO P(*,:,:) :: T
```

The first template dimension is distributed onto the second processor dimension and the third template dimension is distributed onto the third processor dimension.

■

Example: Standard distribution

```
      PROGRAM DIST
CSVM$ PROCESSORS:: P(10), Q(10,10)
CSVM$ TEMPLATE:: T(95,100), T1(100,10)
        ...
CSVM$ DISTRIBUTE (BLOCK,*) ONTO P :: T
CSVM$ DISTRIBUTE (CYCLIC,*) ONTO Q(6:,2) :: T1
        ...
      END
```

The distribution of T is:

```
T(1:10,1:100)  --> P(1)
T(11:20,1:100) --> P(2)
                ...
T(91:95,1:100) --> P(10)
```

The resulting template distribution is slightly unbalanced due to the definition of the `BLOCK` strategy in HPF. The second template dimension is sequentialized. The distribution of *T1* is:

```
T1(1:100:5,1:10)  --> Q(6,2)
T1(2:100:5,1:10)  --> Q(7,2)
                  ...
T1(5:100:5,1:10)  --> Q(10,2)
```

■

Example: General-block distribution

```
      PROGRAM GBDIST
      PARAMETER ( MXNNP=9216 )
CSVM$ PROCESSORS:: P(4)
CSVM$ TEMPLATE:: NODES(MXNNP)
```

```
      INTEGER WORK(4), I
CSVM$ PRIVATE:: I

        ...
      WORK(1) = 1000
      WORK(2) = 4000
      WORK(3) = 4000
CSVM$ REDISTRIBUTE (GENERAL_BLOCK(WORK)) ONTO P :: NODES
CSVM$ PDO(LOOPS(I),STRATEGY(ON_HOME(NODES(I))))
      DO I = 1, MXNNP
          ...
      ENDDO
      END
```

If it is known in advance that there is more work to be done on some iterations than on others, better load balancing can be obtained using the GENERAL_BLOCK distribution format. In contrast to the block strategy where the block size is constant, here each block can vary in size. In this example, the first processor executes 1000 iterations, the second and the third 4000 iterations, and the last processor the rest.

■

Example: RESHAPE and DISTRIBUTE

```
       PROGRAM RESH_DIST
CSVM$ PROCESSORS:: P(8,8) RESHAPE:: Q(64)
CSVM$ TEMPLATE:: T1(100,16), T2(1024)
        ...
CSVM$ DISTRIBUTE (CYCLIC,BLOCK) ONTO P :: T1
CSVM$ DISTRIBUTE (BLOCK) ONTO Q :: T2

        ...
      END
```

Without the RESHAPE attribute of the PROCESSORS directive it would be impossible to distribute the one-dimensional template *T2* onto the whole processor set specified by *P*. Since it is neccessary that they have the same rank, *T2* could only be distributed to the first or the second dimension of *P*, thus only onto 8 processors. Reshaping *P* with the one-dimensional processor set *Q* allows the distribution *T2* onto all the available 64 processors.

■

indirect distribution:

The indirect distribution strategy allows to distribute template elements individually depending on run-time information. The strategy is very flexible since arbitrary integer expressions can be used to determine the processor onto which an element is mapped.

The index variables are local to the directive. The range of values for each variable is the index space of the appropriate template dimension.

The *processor-region* shall consist of a processor arrangement name and index expressions that can be:

- range

  The range expressions shall not reference the index variables and shall not depend on the evaluation of any other expression in the directive.

- integer expressions

  The expressions are arbitrary integer expressions that can contain function references as well as array references. The integer expressions shall not reference more than one index variable. Index variables shall be referenced at the most in one index expression.

The processors to which a template element is mapped are determined by evaluating the index expression of the processor specification. If a section of processors is specified the template element is replicated onto these processors. If an index variable is not referenced in the processor specification the template dimension is sequentialized. A template dimension is also sequentialized if its index variable is used in a sequentialized dimension of the target template.

Example: Indirect distribution

```
      PROGRAM IDIST
      PARAMETER ( MXNNP=9216 )
CSVM$ PROCESSORS:: P(8)
CSVM$ TEMPLATE:: NODES(MXNNP)
      INTEGER MAP(MXNNP), I

        ...
      READ(*,*) MAP

CSVM$ REDISTRIBUTE (I) ONTO P(MAP(I)) :: NODES

       ...
      END
```

In case of an irregular distribution, it is possible to read the mapping from disk and to distribute a template accordingly. The values of *MAP* determine the processor indices of *P* to which the corresponding elements of the template *NODES* are to be mapped. Here, these values range from 1 to 8. An integer variable in the *dist-format* introduces a local variable that can appear

in the *dist-spec-list* again, in combination with other integer variables. The semantics of this specification are those of an implied DO-loop, e.g. *I* covers the declared range of *NODES*. The difference to the GENERAL_BLOCK distribution is that the iterations assigned to a processor don't have to be adjacent to each other.

■

linked distribution:

A template *T* is distributed according to the distribution of another template *T1* via a linked distribution. The HOME clause is used to describe the relation to another template. Note that, if *T1* is redistributed at run-time, *T* is not redistributed automatically. It has to be redistributed explicitly.

The index variables are local to the directive. The range of values for each variable is the index space of the appropriate template dimension of *T*.

The following alignment specifications are possible:

- integer expressions

  The integer expression shall not reference more than one index variable. Index variables shall be referenced at the most in one index expression.

- ':'

  While an integer expression gives a single index value the ':' represents all index values in that template dimension.

The processors to which a template element of *T* is assigned are those processors to which the resulting template elements of *T1* are assigned.

To reduce the run-time overhead of this distribution strategy we allow a slightly larger set of processors in the case of a ':' alignment specification. If a template dimension of *T1* with a ':' alignment specification is distributed to a range of a processor arrangement dimension the template element of *T* is replicated to the entire range.

Example: Linked distribution

```
      PROGRAM LDIST
CSVM$ PROCESSORS:: P(NUMPROC())
      INTEGER NPCB(1000)
CSVM$ TEMPLATE:: NODES(9216)
CSVM$ TEMPLATE:: CAUCHY_NODES(1000)

        ...
CSVM$ DISTRIBUTE (BLOCK) ONTO P :: NODES
CSVM$ REDISTRIBUTE (I) ONTO HOME(NODES(NPCB(I)))::CAUCHY_NODES
        ...
      END
```

In this example, taken from a finite element code, *NODES* is distributed in blocks onto *P*. A subset of the nodes are Cauchy nodes and *NPCB* holds the global node number for each Cauchy node. Thus it is a good policy to distribute *CAUCHY_NODES* according to the distribution of *NODES*.

■

Example: Replication in Linked Distribution

```
CSVM$ PROCESSORS:: P(3,4)
CSVM$ TEMPLATE::   T(2,2), T1(2,2)
CSVM$ DISTRIBUTE (BLOCK,BLOCK) ONTO P(1:3,1:3)::T
CSVM$ DISTRIBUTE (I,J) ONTO HOME(T(I,*)):: T1
```

The distribution of T is:

```
T(1,1) --> P(1,1)
T(1,2) --> P(1,2)
T(2,1) --> P(2,1)
T(2,2) --> P(2,2)
```

The distribution of T1 is:

```
T1(1,1) --> P(1,1:3)
T1(1,2) --> P(1,1:3)
T1(2,1) --> P(2,1:3)
T1(2,2) --> P(2,1:3)
```

This example demonstrates the replication via linked distribution. Although no template element of *T* is mapped to *P(1,3)* and *P(2,3)* the elements of *T1* are replicated to these processors. The reason is that *T* was mapped onto the processor region *P(1:3,1:3)*. The template elements of *T1* are not replicated to *P(1:4,4)* for the same reason. The explicit specification of the processor region determines the replication.

■

7.5 Automatic Template Distribution

Syntax:

undef-directive **is** `UNDEF::` *template-name-list* ◇

Description:

Templates can be distributed automatically according to a dynamic scheduling decision taken for a parallel `DO`-loop. This is specified by the `DISTRIBUTE_ONCE` option.

The `UNDEF`-directive is an executable statement and is used to undefine a template's distribution. Templates with an undefined distribution are distributed when `DISTRIBUTE_ONCE` is used in the `STRATEGY`-option of the `PDO`-directive. The distribution is determined by the schedule of the iterations. The next time a `PDO` is encountered the template's distribution is no longer undefined and the distribution is used to schedule the loop.

An `UNDEF` directive has to be executed by all processors in the processor set that was active when the template was declared. The operation is enclosed in barrier synchronizations.

Example: Semi-static loop scheduling

```
      PROGRAM UNDEF
CSVM$ PROCESSORS:: P(4)
CSVM$ TEMPLATE:: T(100)
         ...
      DO I = 1, 10
CSVM$   PDO(LOOPS(J),STRATEGY(DISTRIBUTE_ONCE(T(J)),(SELF))
        DO J = 1, 100
           ...
        ENDDO
      ENDDO
CSVM$ UNDEF:: T
         ...
      END
```

In the first iteration of the sequential *I*-loop the distribution of *T* is undefined. Thus the strategy used to schedule the parallel *J*-loop is self scheduling. Since *T* is undefined the schedule is stored in *T*. In the following iterations of the *I*-loop the distribution of *T* is defined and consequently used to schedule the parallel loop.

After the last iteration of the *I*-loop, the template's distribution is set undefined such that when the loop is visited again a new work distribution is computed.

■

8 SVM Support

Syntax:

prefetch-directive	**is**	PREFETCH (*access-mode* (*memory-region-list*))
access-mode	**is**	READ
	or	WRITE

◇

Description:

The directive gives a hint to the compiler to prefetch memory regions. With the READ- or WRITE-option it is possible to distinguish the mode in which the memory region will be accessed.

Example: Prefetching

```
        PROGRAM prefetch
CSVM$   PDO(LOOPS(i))
        DO i=1,n
           ...
CSVM$      PREFETCH(READ(c(i)))
CSVM$      PREFETCH(WRITE(a(i)))
           ...
           a(i) = c(i)
           ...
        ENDDO
        END
```

■

9 Intrinsic Functions

Syntax:

intrinsics	**is**	`NUMPROC()`
	or	`NUMPROCSET()`
	or	`MYPROC()`
	or	`MYPROCSET()`
	or	`IPAGESIZE()`
	or	`IPROCID(` *processors-name* [*int-expr-list*]
	or	`MYPROCIND(` *processors-name, dimension, variable-name*)
dimension	**is**	*int-expr*

◇

Description:

The `NUMPROC`-function returns the number of processors available to the program i.e. the number of processors in the initial processor set.

The `NUMPROCSET`-function returns the number of processors belonging to the active processor set.

The `MYPROC`-function returns the global processor number for the asking node. Counting starts with 0 and ends with `NUMPROC`()-1.

The `MYPROCSET`-function returns the processor number in the active processor set to which the asking processor belongs. Counting starts with 0 and ends with `NUMPROCSET`()-1.

The `IPAGESIZE`-function returns the size of a page (the unit of coherence) in bytes.

The `IPROCID`-function returns the global processor number for a processor arrangement member.

The `MYPROCIND`-function returns the index of the asking processor in the dimension specified with *dimension* of the processor arrangement *processor-name*. The success or failure of the operation is returned in the variable *variable-name* which shall be of integer type. If the value of this variable is 1 the processor was found in that dimension; otherwise the variable value is 0.

Example: Usage of intrinsic functions

```
        PROGRAM intrinsics
CSVM$   DEFAULT(PRIVATE)
CSVM$   PROCESSORS::  P(-4:4)
        WRITE(*,*) NUMPROC(), NUMPROCSET(),
     +             MYPROC(), MYPROCSET()
        isize = IPAGESIZE()
        id = IPROCID(P(-4))
        myindex = MYPROCIND(P,1,IERR)
        END
```

The function call *IPROCID(P(-4))* returns 0 as *P(-4)* is mapped to the first processor of the initial processor set. The function call *MYPROCIND(P,1,IERR)* returns the index -4 to 4 for processors numbered 0 to 8. ■

Example: Loop-level scheduling for nested parallel loops

```
        PROGRAM using_myprocind
CSVM$   PROCESSORS:: P(8,4)
CSVM$   PDO (LOOPS(i), PROCESSORS(P(:,*)),STRATEGY(BLOCK))
        DO i=1,100
                ...
CSVM$      PDO (LOOPS(j),
CSVM$+          PROCESSORS(P(MYPROCIND(P,1,IERR)),*),
CSVM$+          STRATEGY(BLOCK))
           DO j=1,100
              ...
           ENDDO
              ...
        ENDDO
        END
```

Each iteration of the outer loop is replicated onto 4 processors. The iterations of the inner loop are distributed onto these 4 processors. ■

Index